System programming (03KTOOT)

Laboratory #4: Kernel and Drivers

Laboratories are mandatory. Each student must submit the work (exclusively in .zip format, with text in .pdf and code (if required) in a format able to be run on the LABINF computers) following the rules published on the web portal. Please remember that for each laboratory the final mark can get from -2.0 (not submitted or with severe flaws) up to +0.5 points (excellent work). If all 4 laboratories are not submitted 8 points are lost.
The deadline for the fourth laboratory is Jan 22nd 2016 at 7 p.m.

Exercise 1: Download 3.3.x version of the Linux kernel and decompress it. Configure the new kernel, justifying the choices. Don't forget to save the running configuration file (in order to be able to replicate the laboratory at home). Compile and install the new kernel, paying attention to preserve the old running kernel. Verify the grub configuration as seen in the slides. Download exercises of the book "Writing Linux Device Drivers" Jerry Cooperstein ISBN 978-1448672387, from: http://www.coopj.com/LDD. Copy and verify the 32 bit kernel configuration file, adjusting it if required. Finally compile and install the kernel. Do not forget that the compliant Ubuntu version for the task is the 12.x LTS.

Exercise 2: Using the schema provided at the end of the document, complete the LDD (and its Makefile):

1. Open and Close functions need full code implementation. Try to avoid multiple instances of the LDD.
2. Init and Exit functions need additional code to properly handle the correct registration/unregistration of the device.
3. Properly create the Makefile (*use this exact name without extension!!!*).

Once the implementation is completed, the following instructions allow you to successfully install your LDD (all commands require *superuser* privileges):

1. Create the device (c = char device, 60 = major number, 0 = minor number):

   $ mknod /dev/my_device c 60 0

2. Make the LDD:

   $ make

3. Install the device:

   $ insmod <LDD_name>.ko

4. Check if it is properly working
5. Remove the LDD (notice the missing extention):

   $ rmmod <LDD_name>

In order to check if the LDD is properly working, develop a set of programs able to:

1. Read only from the device.
2. Then write only (look for a way to test if the write works properly).
3. Finally, read and write.

Extra assignment:
When all notions will be clear enough, try to implement the following concurrent program, using the LDD as input/output for processes/threads.

Generates up to 4 *players* (choose if implement them using threads or processes), where 1 of them expects to read from the LDD while the others attempt to update a number from the LDD, flipping a virtual coin to decide, once read the current number in the LDD, if increase or decrease that number by 1.
Be aware that the LDD connection is a shared good among players so eventually protect the access to it.

Makefile

```
ifneq ($(KERNELRELEASE),)
     obj-m += <YOUR LDD FILENAME WITHOUT .C !!!>.o
else

     KERNELDIR ?= /lib/modules/$(shell uname -r)/build
     PWD := $(shell pwd)

default:
          $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
          $(MAKE) -C $(KERNELDIR) M=$(PWD) clean
endif
```

LDD Schema

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

#define DEVICE_NAME            "my_device"
#define MAJOR_DEVICE_NUMBER    60
#define MINOR_DEVICE_NUMBER    0
#define BUF_LEN                1024

static char msg[BUF_LEN];
static char *msg_ptr;
static int major;

MODULE_AUTHOR("<YOUR NAME>");
MODULE_LICENSE("GPL");

static ssize_t my_read (
     struct file *filp, char __user *buf,
     size_t length, loff_t *offset);
static ssize_t  my_write (
     struct file *filp, const char __user *buf,
     size_t length, loff_t *offset);
static int my_open (struct inode *inode,
                    struct file *filp);
static int my_close (struct inode *inode,
                    struct file *filp);
static int __init my_init (void);
static void __exit my_cleanup (void);

static struct file_operations fops =    {
     .read = my_read,
     .write = my_write,
     .open = my_open,
     .release =  my_close,
};

static int  my_open (struct inode *inode,
                    struct file *filp)
{
     return 0;
}

static int my_close (struct inode *inode,
                    struct file *filp)
{
     return 0;
}
```

```
static ssize_t my_read (
     struct file *filp, char __user *buf,
     size_t length, loff_t *offset)
{
int nc = 0;

     // if no more "valid" bytes can be read, stop
     if (*msg_reading_offset == 0) return 0;

     // no-negative values allow
     if (length < 0)
          return -EINVAL;

     // if a bigger number of bytes than the real one, it get only
what it has
     if (length > strlen(msg)) {
          length = strlen(msg);
     }

     nc = copy_to_user(buf, msg_reading_offset, length);

     /*
          updates the current reading offset pointer so that a
recursive call due to not original
          full lenght will get a 0 (nothing to read)
     */
     msg_reading_offset += sizeof(char) * (length-nc);

     // returns the number of REAL bytes read.
     return length - nc;
}

static ssize_t  my_write (
     struct file *filp, const char __user *buf,
     size_t length, loff_t *offset)
{

     int nc = 0;

     if (length > BUF_LEN)
          return BUF_LEN-length;

     nc = copy_from_user(msg,buf,length);
     msg_ptr = msg;

     return length - nc;
}

static int __init my_init (void)
{
     register_chrdev (MAJOR_DEVICE_NUMBER,
                          DEVICE_NAME,
                          &fops);
```

```
}
module_init(my_init);

static void __exit my_cleanup (void)
{
     unregister_chrdev (major, DEVICE_NAME);
}

module_exit(my_cleanup);
```